ELSEVIER

Theoretical Computer Science 290 (2003) 1541–1556

Theoretical Computer Science

www.elsevier.com/locate/tcs

A time-optimal solution for the path cover problem on cographs ☆

Koji Nakano[a,*], Stephan Olariu[b], Albert Y. Zomaya[c]

[a] *School of Information Science, Japan Advanced Institute of Science and Technology, Tatsunokuchi, Ishikawa 923-1292, Japan*

[b] *Department of Computer Science, Old Dominion University, Norfolk, VA 23529, USA*

[c] *Department of Electrical and Electronic Engineering, The University of Western Australia, Perth, WA 6970, Australia*

Received 8 October 2001; received in revised form 11 January 2002

Communicated by O.H. Ibarra

Abstract

We show that the notoriously difficult problem of finding and reporting the smallest number of vertex-disjoint paths that cover the vertices of a graph can be solved time- and work-optimally for cographs. Our result implies that for this class of graphs the task of finding a Hamiltonian path can be solved time- and work-optimally in parallel.

It was open for more than 10 years to find a time- and work-optimal parallel solution for this important problem. Our contribution is to offer an optimal solution to this important problem. We begin by showing that any algorithm that solves an instance of size n of the problem must take $\Omega(\log n)$ time on the CREW, even if an infinite number of processors are available. We then go on to show that this time lower bound is tight by devising an EREW algorithm that, given an n-vertex cograph G represented by its cotree, finds and reports all the paths in a minimum path cover in $O(\log n)$ time using $n/\log n$ processors.

1. Introduction

Graphs are, unquestionably, among the few fundamental objects that arise naturally in many algorithms in computer science and engineering. A graph-theoretic problem

☆ This work was supported in part by NSF grant CCR-9522093, by ONR grant N00014-97-1-0526, and and by the Australian Research Council.

* Corresponding author.

E-mail address: knakano@jaist.ac.jp (K. Nakano).

0304-3975/02/$ - see front matter

PII: S0304-3975(02)00068-3

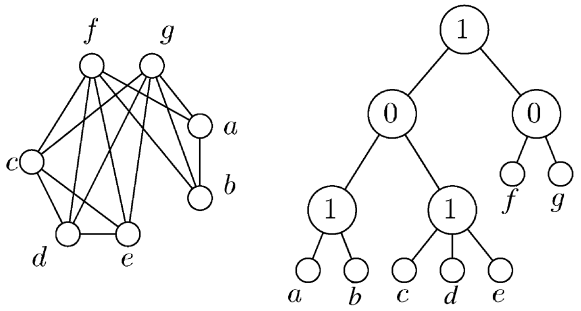

Fig. 1. A cograph and the corresponding cotree.

with a large number of practical applications is the *path cover problem*, which involves finding a minimum number of vertex-disjoint paths that together cover the vertices of a graph. The path cover problem finds application to database design, networking, VLSI design, ring protocols, code optimization, and mapping parallel programs to parallel architectures, among many others [2,8]. It is well known [8] that, as many other important problems in graph theory, the path cover problem and many of its variants are NP-complete.

A graph G that admits a path cover of size one is referred to as *Hamiltonian*. If the unique path that covers G can be extended to a cycle, G is said to possess a Hamiltonian cycle. It is, therefore, clear that the path cover problem is at least as hard as the problem of deciding whether a graph G has a Hamiltonian path (resp. cycle).

Our experience shows that in spite of the fact that many interesting problems are NP-complete on general graphs, in practical applications one rarely has to contend with arbitrary graphs. Typically, a careful analysis of the problem at hand reveals sufficient structure to limit the graphs under investigation to a restricted class.

The class of *cographs*, or complement-reducible graphs is defined recursively as follows:

(1) A single-vertex graph is a cograph;
(2) If $G=(V,E)$ is a cograph, then its complement $\bar{G}=(V, V\times V-E)$ is also a cograph;
(3) If both $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ satisfying $V_1\cap V_2=\phi$ are cograph, then their union $G=(V_1\cup V_2, E_1\cup V_2)$ is also a cograph.

The cographs admit a tree representation unique up to isomorphism. Specifically, one can associate with every cograph $G=(V,E)$ a unique rooted tree $T(G)$ called the *cotree* of G featuring the following properties:

(4) Every internal node of $T(G)$ has at least two children;
(5) The internal nodes of $T(G)$ are labeled by either 0 (0-node) and 1 (1-node) in such a way that labels alternate along every path in $T(G)$ starting at the root;
(6) Each leaf of $T(G)$ corresponds to a vertex in V, such that, $(x,y)\in E$ if and only if the lowest common ancestor of the leaves corresponding to x and y is a 1-node.

We refer the reader to Fig. 1 illustrating a cograph and its cotree. He [12] showed that the cotree can be built from a cograph in $O((\log n)^2)$ time using $O(n+m)$ CRCW processors, where n and m are the number of nodes and edges of the cograph.

In this work we adopt the Parallel Random Access Machine model (PRAM, for short) which consists of synchronous processors, each having access to a shared memory. In each step, the processors perform the same instruction, with a number of processors masked out. In the Concurrent Read Concurrent Write PRAM (CRCW) several processors may simultaneously access the same memory location for both reading and writing; in the Concurrent Read Exclusive Write PRAM (CREW), a memory location can be simultaneously accessed by more than one processor for reading, but not for writing; in the Exclusive Read Exclusive Write PRAM (EREW), a memory location cannot be simultaneously accessed by more than one processor for reading or writing. The interested reader is referred to [13,14] for an excellent discussion of the PRAM family.

Consider a parallel algorithm that solves an instance of size n of some problem in time $T_p(n)$, with p standing for the number of processors used. Traditionally, the main complexity measure for assessing the performance of the algorithm is the *work* $W(n)$ performed, defined as the product $p \times T_p(n)$. The algorithm is termed *work-optimal* if $W(n) \in \Theta(T^*(n))$, where $T^*(n)$ is the running time of the fastest *sequential* algorithm for the problem. Occasionally, an even stronger complexity metric is being used—the so-called *time-optimality*. Specifically, an algorithm is termed time-optimal within a model if no other parallel algorithm solving the same problem and using a polynomial number of processors can run faster in that model, even if an infinite number of processors is available. As it turns out, our solution to the path cover problem on cographs is both work- and time-optimal.

Lin et al. [17] showed that an instance of size n of the path cover problem can be solved in $O(n)$ sequential time. Quite a while back, Adhar and Peng [2] presented a parallel algorithm to find a minimum path cover, a Hamiltonian path, and a Hamiltonian cycle in n-vertex cographs. Their algorithm runs in $O(\log^2 n)$ time and using $O(n^2)$ processors on the CRCW. Surprisingly, the algorithm in [2] takes $O(\log^2 n)$ time using $O(n^2)$ processors on the CRCW even to determine whether a cograph contains a Hamiltonian path or cycle. Adhar and Peng [2] left as an open problem to design a work-optimal algorithm for the path cover problem for cographs.

As a first step towards solving this open problem, Lin et al. [18] showed that one can determine the number of paths in a minimum path cover for cographs in $O(\log n)$ time and $O(n)$ work on the EREW. At the same time, Lin et al. [18] proposed an algorithm to report all the paths in a minimum path cover running in $O(\log^2 n)$ time and using $n/\log n$ processors on the EREW. Thus, the algorithm in [18] is suboptimal.

The main contribution of this work is to offer a time- and work-optimal solution to the graph cover problem for cographs. We begin by establishing a $\Omega(\log n)$ time lower bound on the CREW for the simpler task of determining the number of paths in a minimum path cover. We then go on to show that this lower bound is tight by providing an algorithm that, with an n-vertex cograph G represented by its cograph as input, reports all the paths in a minimum path cover of G in $O(\log n)$ time and $O(n)$ work on the EREW.

Our algorithm uses novel techniques that we outline next. Instead of constructing a minimum path cover directly, we begin by constructing *path trees*, whose inorder traversal corresponds to a minimum path cover. Further, in order to construct the path

trees in parallel, we generate a sequence of brackets that capture the essence of the corresponding cotree. By finding matchings in the set of these brackets we generate the path trees. Finally, the path trees are converted to a minimum path cover for the graph.

The remainder of this work is organized as follows. Section 2 provides background information on the minimum path cover problem and reviews a number of results that will be used as stepping stones in the remainder of this work. Section 3 introduces path trees and establishes their relevance to the minimum path cover problem. Section 4 shows that for cographs, path trees can be obtained fast by using bracket matching. Section 5 presents the details of our optimal algorithm for minimum path cover for cographs Finally, Section 6 provides concluding remarks and open problems.

2. Finding a minimum path cover: a first look

We begin this section by establishing a $\Omega(\log n)$ time lower bound for the tasks of

- determining the number of paths in a minimum path cover for an n-vertex cograph represented by its cotree, and
- returning the paths in a minimum path cover of an n-vertex cograph represented by its cotree.

For this purpose, we reduce the well known OR problem to the two problems above. Recall that the OR problem, given a sequence of n bits asks to determine their logical OR. For further reference, we now state a fundamental result of Cook et al. [6] that will lay the foundation of our time lower bound argument.

Lemma 2.1. *The time lower bound for computing the OR of n bits on the CREW is* $\Omega(\log n)$ *even if an infinite number of processors and memory cells are available.*

Let $b_1, b_2, \ldots, b_n$ be an arbitrary input to the OR problem. From this input we construct a cotree $T(G)$ as follows. The cotree $T(G)$ has two internal nodes R and u: R is the root labeled by 0, and u is its child labeled by 1. With every bit b_i, $(1 \leqslant i \leqslant n)$, we associate leaf a_i. Each leaf a_i, $(1 \leqslant i \leqslant n)$, is connected to R if $b_i = 0$, and is connected to u, otherwise. Further, to satisfy (4), we use three leaves x, y, and z: x is connected to R, and y and z are connected to u. Clearly, both R and u always have two children and so (4) is satisfied. It is trivial to construct the cotree $T(G)$ using the well-known parent-pointer representation. Furthermore, it is easy to confirm that n CREW processors can complete the construction $T(G)$ in O(1) time. If $b_i = 1$ then a_i belongs to a path containing y. Otherwise, a_i is an isolated vertex in G and is itself a path in a minimum path cover of G. The reader is referred to Fig. 2 for an illustration of this construction corresponding to the bit sequence $0, 0, 0, 0, 0, 1, 0, 1$. Clearly, if the input $b_1, b_2, \ldots, b_n$ has k 1's, then, the path containing y has $k+2$ vertices and a minimum path cover has $n-k+2$ paths. Therefore, the input has at least one 1, iff

- the path containing y has more than two vertices, and iff
- the number of paths in a minimum path cover is less than $n+2$.

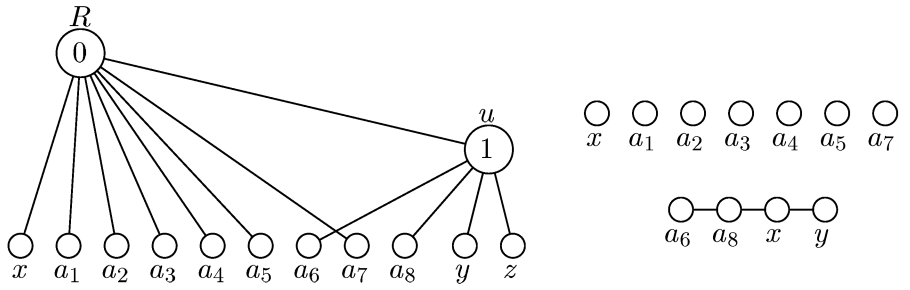

Fig. 2. Illustrating the time lower bound argument.

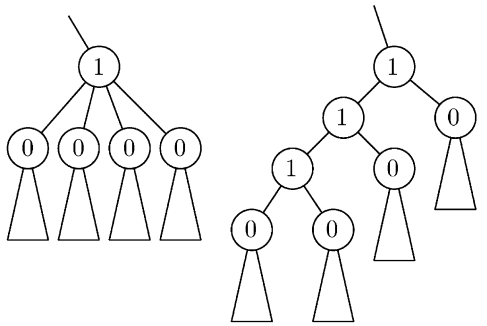

Fig. 3. Illustrating the binarization process.

Now, Lemma 2.1 guarantees that $\Omega(\log n)$ must be a time lower bound for the problem of determining the number of paths in a minimal path cover of an n-vertex cograph and of reporting all paths in a minimum path cover. To summarize, we have proved the following result:

Theorem 2.2. *Every algorithm that determines the number of paths in a minimum path cover or reports a minimum path cover on an n-vertex cograph represented by its cotree must take* $\Omega(\log n)$ *CREW time even if an infinite number of processors is available.*

Next, we review the sequential algorithm of [17] for finding a minimum path cover of n-vertex cograph in $O(n)$ time, as well as the parallel algorithm of [18] for computing the number of paths in a minimum path cover in $O(\log n)$ time using $n/\log n$ processors on the EREW.

For convenience and ease of presentation we now show how to binarize the cotree $T(G)$ corresponding to a cograph G, in such a way that each of its internal nodes has exactly two children. Let u be an internal node with children $v_1, v_2, \ldots, v_k$, $(k \geqslant 3)$, and refer to Fig. 3. We replace node u by $k-1$ nodes $u_1, u_2, \ldots, u_{k-1}$ such that u_1 has children v_1 and v_2, and each u_i, $(2 \leqslant i \leqslant k)$, has children u_{i-1} and v_i. We shall refer

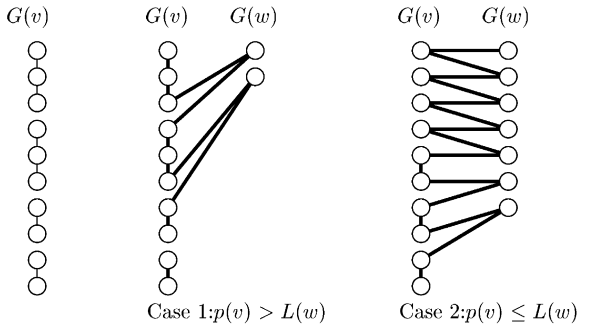

Fig. 4. Illustrating Cases 1 and 2.

to the binarized version of $T(G)$ as $T_b(G)$ and note that $T_b(G)$ satisfies properties (4) and (6) above.

For an internal node u of $T_b(G)$ its *left* and *right* children will be denoted by v and w, respectively. Let $G(u)$ denote the subgraph of G induced by the leaf descendants of u in $T_b(G)$. Further, let $L(u)$ denote the *number* of leaf descendants of u in $T_b(G)$, that is, the number of vertices of $G(u)$. Let $p(u)$ denote the number of paths in the minimum path cover of $G(u)$. We say that $T_b(G)$ is *leftist*, if for every internal node u, the condition $L(v) \geqslant L(w)$ is satisfied. Let $T_{bl}(G)$ denote the leftist binarized cotree of G.

We now review the ideas developed in [17] for finding a minimum path cover of a cograph G, given its leftist binarized cotree $T_{bl}(G)$.

Suppose that the minimum path covers of $G(v)$ and $G(w)$ have already been obtained. If u is 0-node, then no edge in $G(u)$ connects vertices from $G(v)$ and $G(w)$. Thus, a minimum path cover for G is just the union of minimum path covers for $G(v)$ and $G(w)$.

If u is 1-node, recall that every vertex in $G(v)$ is adjacent to all the vertices in $G(w)$. Referring to Fig. 4, we distinguish the following two cases:

Case 1: $p(v) > L(w)$. We use the $L(w)$ vertices in $G(w)$ to bridge $L(w)+1$ of the paths in a minimum path cover of $G(v)$ into one path and the resulting minimum path cover has $p(v) - L(w)$ paths. In Fig. 4, $L(w) = 2$ nodes bridge $p(v) = 4$ paths into $p(v) - L(w) = 2$ paths.

Case 2: $p(v) \leqslant L(w)$. In this case, $p(v) - 1$ vertices in $G(w)$ are used to bridge the $p(v)$ paths in a minimum path cover of $G(v)$ into one path. These vertices are said to be *bridge* vertices. The remaining $L(w) - p(v) + 1$ vertices, called *insert* vertices, will be inserted into the path thus obtained. The resulting minimum path cover is a Hamiltonian path. In Fig. 4, $p(v) - 1 = 3$ nodes are used to bridge $p(v) = 4$ paths into one path and the $L(w) - p(v) + 1 = 4$ nodes are inserted into the path.

We refer the reader to [17] for a detailed proof of the correctness of this approach. As it turns out, all the paths in a minimum path cover for G can be obtained by traversing $T_{bl}(G)$ in a bottom-up fashion from the leaves to the root. A careful implementation guarantees that the corresponding algorithm runs in time linear in the size of $T_{bl}(G)$.

Thus, we have the following result.

Lemma 2.3 (Lin et al. [17]). *Given the leftist binarized cotree $T_{bl}(G)$ of an n-vertex cograph G, a minimum path cover can be returned in $O(n)$ sequential time.*

Although the sequential algorithm is quite simple, it seems extremely hard to parallelize. Let us estimate the complexity of a naive parallelization. The construction of a minimum path cover corresponding to a 1-node needs $O(\log n)$ time. Thus, the naive parallelization needs $O((\text{height of } T_{bl}(G)) \times \log n)$ time to report all the paths in a minimum path cover. Notice that, in the worst case, the height of $T_{bl}(G)$ is $O(n)$.

Lin et al. [18] showed that the simpler problem of computing the *number* of paths in the minimum path cover can be solved in $O(\log n)$ time. Specifically, they showed that the number $p(u)$ of paths in given by the following formula:

$$p(u) = \begin{cases} p(v) + p(w) & \text{if } u \text{ is 0-node,} \\ \max\{p(v) - L(w), 1\} & \text{if } u \text{ is 1-node.} \end{cases}$$

The tree contraction [1,13] enables us to evaluate this formula for each internal node u. Using this idea, the following result was proved in [18].

Lemma 2.4. *For every internal node u of $T_{bl}(G)$, the number $p(u)$ of paths in the minimum path cover of $G(u)$ can be computed in $O(\log n)$ time using $n/\log n$ EREW processors.*

We now further modify $T_{bl}(G)$. The vertices of the cograph G (i.e. leaves of $T_{bl}(G)$) will be partitioned into three categories as follows:
bridge vertex a vertex bridging paths at a 1-node;
insert vertex a vertex to be inserted in the path at a 1-node;
primary vertex a vertex neither bridging nor being inserted.
Note that a primary vertex corresponds to a leaf of $T_{bl}(G)$ such that every internal node along a path from the root to the leaf is not the right child of a 1-node. Conversely, a bridge or insert vertex belongs to a subtree rooted at an internal node that is the right child of a 1-node.

For every 1-node u of $T_{bl}(G)$, the structure of the subtree rooted at the right child w is immaterial, because vertices in $G(w)$ are used to bridge or to be inserted and, therefore, the edges in $G(w)$ are never used in the path cover. Using this observation, we can ignore the *structure* of this subtree and replace it by a set of leaves as illustrated in Fig. 5. More precisely, for a 1-node u with left and right children v and w, the subtree rooted at w is replaced by $L(w)$ leaves if w is not itself a leaf. If $p(v) > L(w)$ (see Case 1 above), then all of the $L(w)$ children are bridge vertices. Otherwise, if $p(v) \leqslant L(w)$ (see Case 2 above), $p(v) - 1$ of the children are bridge vertices and the remaining $L(w) - p(v) + 1$ children are insert vertices. For the sake of consistency, w is changed to 0-node. After performing the changes detailed above we obtain the reduced leftist binarized cotree of G, denoted henceforth by $T_{blr}(G)$.

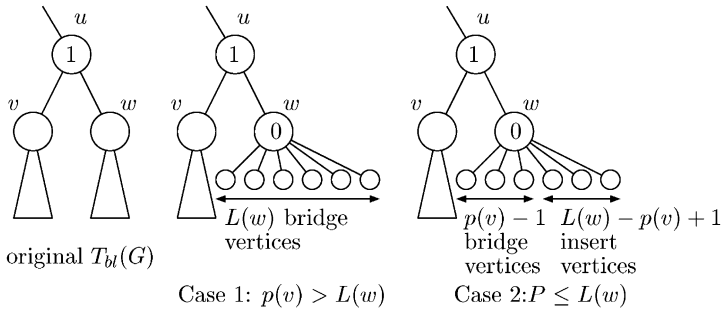

Fig. 5. Illustrating the reduced leftist binarized cotree $T_{\mathrm{blr}}(G)$.

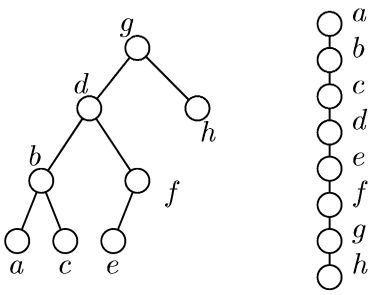

Fig. 6. A path tree and the corresponding path.

3. Finding a minimum path cover using path trees

The main goal of this section is to introduce *path trees* that will turn out to be key ingredients in out time- and work-optimal parallel algorithm for the path cover problem.

Let G be a cograph. A path tree is a rooted binary tree with each node of the path tree corresponding to a vertex of some path π in G. The path π is captured by the inorder traversal of the path tree. We refer the reader to Fig. 6 illustrating a path tree and the corresponding path. Clearly, once a path tree is available it can be readily converted into the desired path by the Euler tour technique. Multiple vertex-disjoint paths will be captured by disjoint collections of path trees.

Let u be an internal node of $T_{\mathrm{blr}}(G)$ with left and right children v and w, respectively. We are interested in computing a path tree of $G(u)$ using those for $G(v)$ and $G(w)$. First, suppose that u is a 0-node and the path trees of $G(v)$ and $G(w)$ are already available. Since no edge in the graph connects edges from $G(v)$ and $G(w)$, the union of the path trees for $G(v)$ and $G(w)$ yields the path trees for $G(u)$.

Next, suppose that u is a 1-node and the path trees of $G(v)$ have already been computed. We consider the following following two cases:

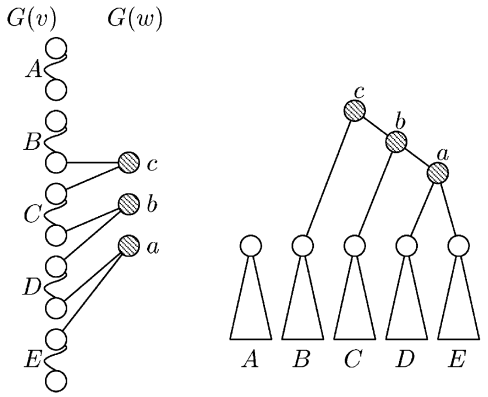

Fig. 7. Illustrating the construction for Case 1: $p(v) > L(w)$.

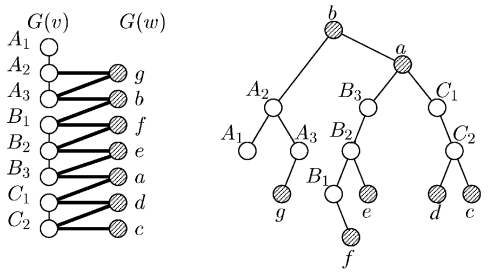

Fig. 8. Illustrating the construction for Case 2: $p(v) \leqslant L(w)$.

Case 1: $p(v) > L(w)$. The $L(w)$ vertices in $G(w)$ bridge $L(w)+1$ paths and the resulting minimum path cover of $G(u)$ has $p(v) - L(w)$ paths. To perform the corresponding operation on the path trees, we first construct a binary tree using the $L(w)$ vertices in $G(w)$. Then, the roots of $L(w)+1$ path trees in $G(v)$ are connected to the leaves of the binary tree thus obtained. This process is illustrated in Fig. 7, where we construct a binary tree with vertices a, b, c having the roots of the path trees B, C, D, E as their children. The inorder traversal of the path tree thus obtained is

$$B \to c \to C \to b \to D \to a \to E,$$

which corresponds to the path we should obtain.

Case 2: $p(v) \leqslant L(w)$. We refer the reader to Fig. 8 for an illustration of the construction of a path tree in Case 2. In this case, $p(v) - 1$ bridge vertices from $G(w)$ connect the roots of the path trees in a way similar to Case 1. In the figure, two vertices a and b connect three path trees. Each of the $L(w) - p(v) + 1$ insert vertices is connected to path trees as leaves. In the figure, five vertices c, d, e, f, and g are connected to path trees as leaves.

Notice that in this process, a vertex of the original path trees with at most on child may end up with one (or two) insert vertices from $G(w)$ as leaves. However, not

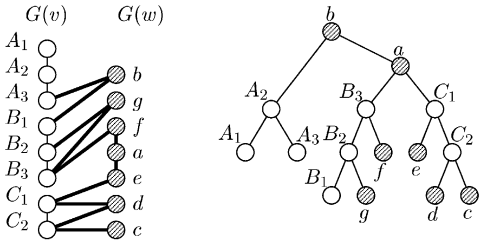

Fig. 9. Illustrating a pseudo path tree.

all such vertices can have a child. For example, C_1 cannot have an insert vertex as a left child; if vertex g were a left child of C_1, then a and g would be adjacent in the corresponding Hamiltonian path. However, it is not necessarily the case the a and g are adjacent in the underlying graph G. For the same reason, B_3 cannot have an insert vertex as a right child. More generally, we may have *illegal* children as follows: Let $T_1, T_2, \ldots, T_{p(v)}$ denote the path trees placed left-to-right in this order such that their roots are connected by $p(v)-1$ bridge vertices.

1. The right child of the rightmost vertex (i.e. the vertex that appears last in the inorder traversal) of T_i, $(1 \leqslant i \leqslant p(v)-1)$. If an insert vertex is connected as the right child, then it must be adjacent to the bridge vertex that is the lowest common ancestor of T_i and T_{i+1}, a contradiction.
2. The left child of the leftmost vertex in T_i, $(2 \leqslant i \leqslant p(v))$, by a mirror argument.

Hence, we have at most $2p(v)-2$ illegal children. For later reference, we introduce a *pseudo path tree*, which may have illegal insert vertices. Fig. 9 illustrates a pseudo path tree: vertices e and f are illegal, and the corresponding path is *invalid* in the graph since the edges (f,a) and (a,e) are not present.

4. Constructing path trees using brackets

As we mentioned in Section 2, the naive parallelization of the linear-time path cover algorithm in [17] takes O((height of $T_{\mathrm{blr}}(G)) \times \log n$) time. For the same reason, if we construct path trees from the leaves of the cotree up to the root, it still needs O((height of $T_{\mathrm{blr}}(G)) \times \log n$) time.

The main goal of this section is to show that we can remove a factor of O(height of $T_{\mathrm{bl}}(G)$) from the running time of the naive implementation. The main vehicle for achieving this goal is the efficient construction of path trees from brackets.

To begin, we demonstrate how pseudo path trees can be constructed efficiently. Once this is done, we show how to convert pseudo path trees to correct path trees. For constructing pseudo path trees efficiently, we will generate a sequence of brackets, each corresponding to a vertex in the pseudo path tree. We use two types of brackets: *square brackets* "[" and "]" and *round brackets* "(" and ")". By finding matching pairs of square brackets and matching pairs of round brackets independently, we can construct pseudo path trees. The details are spelled out as follows.

Let u be a node of $T_{\mathrm{blr}}(G)$ and let v and w denote, respectively, its left and right children if any. We associate with u a sequence $B(u)$ of brackets as follows: If u is a leaf corresponding to a primary vertex, then

$$B(u) = \overset{u^p\ u^{\mathrm{l}}\ u^{\mathrm{r}}}{[\ (\ (}\ .$$

If u is 0-node then

$$B(u) = B(v) \cdot B(w) \quad \text{i.e. the concatenation of } B(v) \text{ and } B(w)$$

If u is a 1-node and $p(v) > L(w)$ (i.e. Case 1), then

$$B(u) = B(v) \cdot \overset{s_1^{\mathrm{r}}\ s_1^{\mathrm{l}}\ s_1^p\ s_2^{\mathrm{r}}\ s_2^{\mathrm{l}}\ s_2^p \quad\quad s_{L(w)}^{\mathrm{r}}\ s_{L(w)}^{\mathrm{l}}\ s_{L(w)}^p}{]\ ]\ [\ ]\ ]\ [\cdots\ ]\ \ ]\ \ [}\ ,$$

where, s_i $(1 \leqslant i \leqslant L(w))$ denotes the bridge vertices of w. For i, $1 \leqslant i \leqslant L(w) - 1$, the matching pair $\overset{s_i^p}{[}$ and $\overset{s_{i+1}^{\mathrm{r}}}{]}$ corresponds to an edge connecting the right child s_i and its parent s_{i+1}. Further, each of $\overset{s_1^{\mathrm{l}}}{]}, \overset{s_2^{\mathrm{l}}}{]}, \ldots, \overset{s_{L(w)}^{\mathrm{l}}}{]}$ and $\overset{s_1^{\mathrm{r}}}{]}$ matches a square bracket in $B(v)$, and corresponds to an edge connecting with the root of a path tree of $G(v)$.

If u is a 1-node and $p(v) \leqslant L(w)$ (i.e. Case 2), then

$$B(u) = B(v) \cdot \overset{s_1^{\mathrm{r}}\ s_1^{\mathrm{l}}\ s_1^p\ s_2^{\mathrm{r}}\ s_2^{\mathrm{l}}\ s_2^p \quad\quad s_{p(v)-1}^{\mathrm{r}}\ s_{p(v)-1}^{\mathrm{l}}\ s_{p(v)-1}^p\ t_{p(v)}^p\ t_{p(v)+1}^p}{]\ ]\ [\ ]\ ]\ [\cdots\ ]\ \ ]\ \ [\ \ )\ \ )}$$
$$\overset{t_{L(w)}^p\ t_{p(v)}^{\mathrm{l}}\ t_{p(v)}^{\mathrm{r}}\ t_{p(v)+1}^{\mathrm{l}}\ t_{p(v)+1}^{\mathrm{r}} \quad\quad t_{L(w)}^{\mathrm{l}}\ t_{L(w)}^{\mathrm{r}}}{\cdots\ )\ \ (\ \ (\ \ (\ \ (\cdots\ (\ \ (}\ ,$$

where, for every i, $(1 \leqslant i \leqslant p(v) - 1)$, s_i denotes a bridge vertex, and for every i, $(p(v) \leqslant i \leqslant L(w))$, t_i denotes an insert vertex. The square brackets for s_i work similarly to Case 1. Each of the round brackets $\overset{t_{p(w)}^p\ t_{p(w)+1}^p \quad\quad t_{L(w)}^p}{)\ \ )\ \cdots\ )}$ is used to find a parent of t_i in $G(v)$. Further, the round brackets $\overset{t_i^{\mathrm{l}}\ t_i^{\mathrm{r}}}{(\ \ (}$ $(p(w) \leqslant i \leqslant L(w))$ are used to find the left and the right children, which will appear to the right of $B(u)$.

By finding matchings of square brackets and round brackets in the sequence $B(R)$ of brackets of the root R of $T_{\mathrm{blr}}(G)$, we can construct pseudo path trees. We refer to the reader to Fig. 10 for an example. The following sequence of brackets corresponds to the cotree illustrated in:

$$\overset{a^p\ a^{\mathrm{l}}\ a^{\mathrm{r}}\ b^p\ b^{\mathrm{l}}\ b^{\mathrm{r}}\ c^p\ c^{\mathrm{l}}\ c^{\mathrm{r}}\ d^{\mathrm{r}}\ d^{\mathrm{l}}\ d^p\ e^p\ f^p\ e^{\mathrm{l}}\ e^{\mathrm{r}}\ f^{\mathrm{l}}\ f^{\mathrm{r}}}{[\ (\ (\ )\ (\ (\ [\ (\ (\ ]\ ]\ [\ \ )\ \ )\ (\ (\ (\ (}\ .$$

Note that a and c are primary vertices, b, e, and f are insert vertices, and d is a bridge vertex. Consider that matching of square brackets and that of round brackets are computed independently. In the above sequence of brackets, we can find the following matching:

$$\overset{a^p\ d^{\mathrm{l}}\ c^p\ d^{\mathrm{r}}\ a^{\mathrm{r}}\ b^p\ c^{\mathrm{l}}\ f^p\ c^{\mathrm{r}}\ e^p}{[\ ],\ [\ ],\ (\ ),\ (\ ),\ (\ )}\ .$$

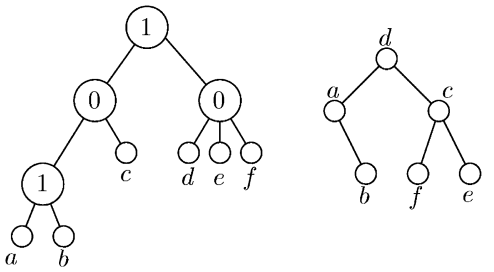

Fig. 10. Construction of a pseudo path tree using brackets.

These matchings correspond to edges of a pseudo path tree. For example, $\overset{a^p\ d^{\mathrm{l}}}{[\ ]}$ corresponds to an edge connecting the vertex a to its parent d as a left child. In general, for vertices a and b, we establish an edge as follows:

$\overset{a^p\ b^{\mathrm{l}}}{[\ ]}$ an edge connecting the vertex a to its parent b as a left child;

$\overset{a^p\ b^{\mathrm{r}}}{[\ ]}$ an edge connecting the vertex a to its parent b as a right child;

$\overset{a^{\mathrm{l}}\ b^p}{(\ )}$ an edge connecting the vertex b to its parent a as a left child;

$\overset{a^{\mathrm{r}}\ b^p}{(\ )}$ an edge connecting the vertex b to its parent a as a right child.

Clearly, since the tree thus obtained may contain illegal insert vertices, it is a pseudo path tree, not a path tree. For example, in Fig. 10, d and f are adjacent in the path, although they may not be connected by an edge in G. In the worst case, we may have up to $2p(v)-2$ illegal insert vertices for each 1-node u with left and right children v and w.

To remove illegal insert vertices, we will use *dummy* vertices that we define next. Consider again a 1-node u with left and right children v and w, respectively, such that $p(v)\leqslant L(w)$. Recall that we may have up to $2p(v)-2$ illegal insert vertices. We add to the pseudo path tree corresponding to $2p(v)-2$ dummy vertices $d_1, d_2, \ldots, d_{2p(v)-2}$ by modifying $B(u)$ as follows:

$$B(u) = B(v) \cdot \overset{s_1^{\mathrm{r}}\ s_1^{\mathrm{l}}\ s_1^p\ s_2^{\mathrm{r}}\ s_2^{\mathrm{l}}\ s_2^p \qquad s^{\mathrm{r}}_{p(v)-1}\ s^{\mathrm{l}}_{p(v)-1}\ s^p_{p(v)-1}\ t^p_{p(v)}\ t^p_{p(v)+1} \qquad t^p_{L(w)}\ d_1^p\ t_2^p \qquad d^p_{2p(v)-2}}{]\ ]\ [\ ]\ ]\ [\cdots\ ]\ \ ]\ \ [\ \ )\ \ )\ \cdots\ )\ \ )\ )\cdots\ )}$$

$$\overset{d_1^{\mathrm{r}}\ e_2^{\mathrm{r}} \qquad d^{\mathrm{r}}_{2p(v)-2}\ t^{\mathrm{l}}_{p(v)}\ t^{\mathrm{r}}_{p(v)}\ t^{\mathrm{l}}_{p(v)+1}\ t^{\mathrm{r}}_{p(v)+1} \qquad t^{\mathrm{l}}_{L(w)}\ t^{\mathrm{r}}_{L(w)}}{(\ (\cdots\ (\ \ (\ \ (\ \ (\ \ (\cdots\ (\ \ (\ .}$$

Here, $\overset{d_i^p}{)}$ and $\overset{d_i^{\mathrm{r}}}{(}$ are used to find a parent and a child for dummy vertex d_i. Notice that, as illustrated in Fig. 11 the leaves of the pseudo path tree of u in a right to left order begin with insert vertices followed by dummy vertices. More specifically, Fig. 11 illustrates $2p(v)-2=4$ dummy vertices d_1, d_2, d_3, and d_4 connected to the

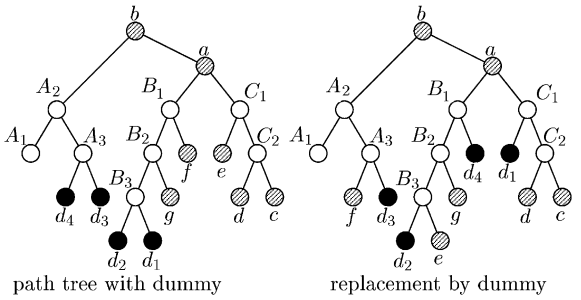

Fig. 11. Construction of a path tree using dummy vertices.

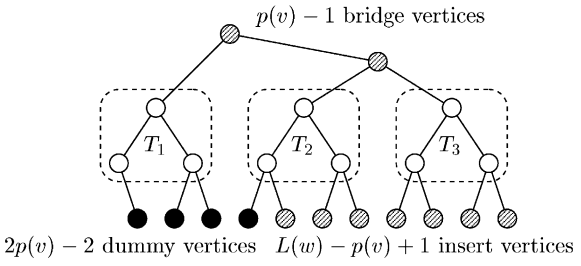

Fig. 12. Illustrating how to place insert and dummy vertices.

pseudo path tree. By construction, dummy vertices can have at most one child. At this point every insert and dummy vertex is checked for legality, that is, whether it is adjacent to a bridge vertex. Illegal insert vertices are exchanged with legal dummy vertices. For example, in Fig. 11, e and f are illegal insert vertices. Also, dummy nodes d_2 and d_3 are illegal. Thus, e and f are exchanged with d_1 and d_4, respectively. Note that this exchanging is not only for the vertices but also for the subtrees. That is, exchanging of e and d_1 means that the parent of d_1 becomes new parent of e and vice versa. After that, bypassing dummy vertices in the path tree, we can obtain a correct path tree.

The readers should have no difficulty to confirm that d_i^p) should find a parent in $G(v)$, and d_i^r (should find a child if any. Since each 1-node u has $2p(v)-2$ dummy vertices and $p(v)-1$ bridge vertices, the total number of dummy vertices is exactly the double of the bridge vertices. Hence, the total number of dummy vertices of an n-vertex cograph is $O(n)$, and thus the sequence $B(R)$ has $O(n)$ brackets.

Let us confirm that by exchanging illegal insert vertices and legal dummy vertices we obtain correct path trees. For a fixed 1-node u with left and right children v and w, suppose that $p(v) \leqslant L(w)$ is satisfied. Let $T_1, T_2, \ldots, T_{p(v)}$ denote the path trees of v enumerated in left to right order, as illustrated in Fig. 12. Let n_i $(1 \leqslant i \leqslant p(v))$ denote the number of non-dummy vertices in T_i. Then, each T_i can connect at most $n_i + 1$

insert and dummy vertices. Thus, the $p(v)$ path trees can connect at most $(n_1+1)+(n_2+1)+\cdots+(n_{p(v)}+1)=L(v)+p(v)$ insert and dummy vertices. Since we have $L(w)-p(v)+1$ insert vertices and $2p(v)-2$ dummy vertices, $(L(w)-p(v)+1)+(2p(v)-2)=L(w)+p(v)-1\leqslant L(v)+p(v)-1$ the $p(v)$ path trees can connect these vertices.

5. The optimal parallel algorithm

The main goal of this section is to provide a matching upper bound for the time lower bound proved in Section 2. Specifically, we will be designing a parallel algorithm for the minimum path cover of a cograph running in $O(\log n)$ time using $n/\log n$ processors in the EREW–PRAM for the input size n. We will show that, for given the cotree $T(G)$ of a cograph G, the minimum path cover can be exhibited efficiently. The algorithm is spelled out as follows:

Input: a cotree $T(G)$ of a cograph G;
Output: a minimum path cover of G;
Step 1: Find the binarized cotree $T_b(G)$;
Step 2: Compute the number $L(u)$ of vertices in $G(u)$ for each internal node u of $T_b(G)$ and compute the leftist binarized cotree $T_{bl}(G)$;
Step 3: Compute the number $p(u)$ of paths in the minimum path cover of $G(u)$ for each internal node u of $T_{bl}(G)$ and compute the reduced leftist binarized cotree $T_{blr}(G)$;
Step 4: Generate the sequence of brackets $B(R)$ of the root R of $T_{blr}(G)$;
Step 5: Find the pseudo path tree by finding all matchings of $B(R)$;
Step 6: Find all the illegal insert vertices in the pseudo path trees and exchange them with legal dummy vertices;
Step 7: Remove dummy vertices to get the correct path trees;
Step 8: Return a minimum path cover from the path trees.

The reader should have no difficulty to confirm that the algorithm above correctly returns a minimum path cover for G. Turning to the complexity, we now show that this algorithm can be implemented to run in $O(\log n)$ time using $n/\log n$ EREW processors.

For this purpose, the next two lemmas summarize key results that we will need in our analysis. See [3,5,9,13] for details.

Lemma 5.1. *Each of the following tasks can be performed in* $O(\log n)$ *time using* $n/\log n$ *processors on the EREW.*

1. *Given a linked list of n vertices, compute the rank (i.e. the distance to the tail of the list) of each vertex in the list;*
2. *Given n integers* $a(1), a(2), \ldots, a(n)$ *in an array, compute all the prefix sums* $a(1)+a(2)+\cdots+a(i)$ *for all* i;
3. *Given a sequence of n brackets, find all the matching pairs in the sequence.*

Further, it also relies on the following algorithms using Euler tour technique [19] with the algorithms for Lemma 5.1.

Lemma 5.2. *Given the adjacency list of a tree T and a specified root vertex R, each of the following tasks can be done in* $O(\log n)$ *time using* $n/\log n$ *processors on the EREW.*

1. Compute the Euler tour of T.
2. Give preorder, postorder, and inorder numbers of each vertex u in T.
3. The number of descendants and the number of descendant leaves of each internal vertex u of T.

Next we discuss the implementation details of our parallel algorithm. Suppose the adjacency list of the cotree $T(G)$ is given. As shown in [1] the task of binarizing $T(G)$ in Step 1 can be performed optimally. By Lemma 5.2, Steps 2 can be performed optimally. By Lemma 2.4, Step 3 can be completed in $O(\log n)$ time using $n/\log n$ EREW processors. The task of converting to bracket representation in Step 4 amounts to computing inorder numbering which, by Lemma 5.2 can be performed optimally. Since the length of the brackets sequence is $O(n)$, Step 5 can be done optimally by Lemma 5.1. The task of finding all illegal vertices can be performed by traversing a path trees in inorder and checking vertex adjacencies in the resulting linear order. By Lemma 5.2, this can be done optimally. In Step 6, exchanging illegal insert vertices and legal dummy vertices can be done optimally by parentheses matching as follows: Each legal dummy vertex is assigned a "(", and every illegal insert vertex is assigned a ")". The task of bypassing dummy vertices in Step 7 is an instance of path compression, a particular case of tree contraction and can be optimally by the algorithm of [1] Finally, Step 8 is an instance of computing Euler tours. Thus we have proved the following result.

Theorem 5.3. *The task of returning a minimum path cover of a cograph can be performed time- and work-optimally in* $O(\log n)$ *time using* $n/\log n$ *processors on the EREW.*

6. Conclusions and open problems

It was open for more than 10 years to find a time- and work-optimal parallel solution for this important problem. Our main results was to show that such an optimal algorithm exists. Specifically, we have shown that given an n-vertex cograph G represented by its cotree as input, a minimum path cover for G can be returned in $\Theta(\log n)$ time using $n/\log n$ EREW processors.

Our solution relied on a number of interesting and novel techniques. It would be important to know if these techniques can be applied to solving additional algorithmic problems for particular graph classes. This is a promising area for further investigations.

References

[1] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, T. Przytycka, A simple parallel tree contraction algorithm, J. Algorithms 10 (1989) 287–302.
[2] G.S. Adhar, S. Peng, Parallel algorithm for path covering, Hamiltonian path, and Hamiltonian cycle in cographs, Proc. Internat. Conf. on Parallel Processing, Vol. III, 1990, pp. 364–365.
[3] R.J. Anderson, G.L. Miller, Deterministic parallel list ranking, Algorithmica 6 (1991) 859–868.
[4] J.A. Bondy, U.S.R. Murty, Graph Theory with Applications, North-Holland, Amsterdam, 1976.
[5] R. Cole, U. Vishkin, Approximate parallel scheduling, Part I: the basic technique with applications to optimal parallel list ranking in logarithmic time, SIAM J. Comput. 17 (1988) 128–142.
[6] S.A. Cook, C. Dwork, R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, SIAM J. Comput. 15 (1986) 87–97.
[7] D.G. Corneil, H. Lerchs, L. Stewart Burlingham, Complement reducible graphs, Discrete Appl. Math. 3 (1981) 163–174.
[8] M.R. Garey, D.S. Johnson, Computers and Intractability, A Guide to the Theory of NP-completeness, Freeman, San Francisco, 1979.
[9] A. Gibbons, W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, Cambridge, 1989.
[10] A. Gibbons, W. Rytter, Optimal parallel algorithms for dynamic expression evaluation and context-free recognition, Inform. Comput. 81 (1989) 32–45.
[11] X. He, Efficient parallel algorithms for solving some tree problems, Proc. 24-th Allerton Conf. on Communication, Control, and Computing, 1986, pp. 777–786.
[12] X. He, Parallel algorithm for cograph recognition with applications, J. Algorithms 15 (2) (1993) 284–313.
[13] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, Reading, MA, 1992.
[14] R.M. Karp, V. Ramachandran, A survey of parallel algorithms for shared memory machines, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, North-Holland, Amsterdam, 1990, pp. 869–941.
[15] C.P. Kruskal, L. Rudolph, M. Snir, Efficient parallel algorithms for graph problems, Algorithmica 5 (1990) 43–64.
[16] R. Lin, S. Olariu, A fast recognition algorithm for cographs, J. Parallel Distrib. Comput. 13 (1991) 76–91.
[17] R. Lin, S. Olariu, G. Pruesse, An optimal path cover algorithm for cographs, Comput. Math. Appl. 30 (1995) 75–83.
[18] R. Lin, S. Olariu, J.L. Schwing, J. Zhang, A fast EREW algorithm for minimum path cover and hamiltonicity for cographs, Parallel Algorithms Appl. 2 (1994) 99–113.
[19] R.E. Tarjan, U. Vishkin, An efficient parallel biconnectivity algorithm, SIAM J. Comput. 14 (1985) 862–874.